IBM Employee Churn Prediction

In this project, I explain why employees choose to exit firms, by analyzing data on employee and job characteristics and how they drive exit decisions. I first use linear probability model and a logistic regression model to estimate the contribution of each characteristic to exit decisions. This helps me evaluate which features are most important in driving exits. Next, I use 8 different types of classifier models to predict exit decisions based on a training set, and then evaluate these models on a test set.

Executive Summary

(A) Project Goal:

1. Explain effects of employee and job characteristics on exit decision
2. Predict employee churn (exit) using labelled data on employee and job characteristics

(B) Data Analyses Methods (in order of application):

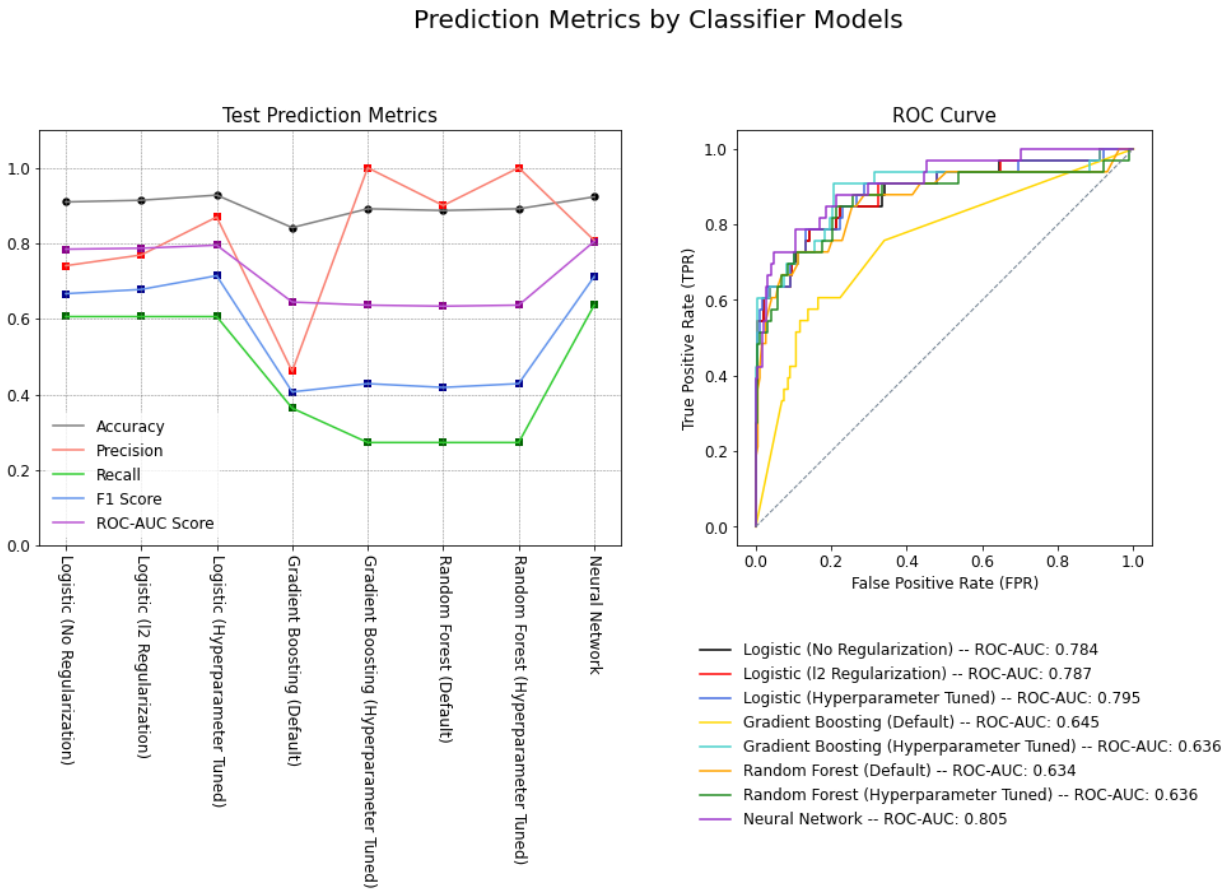
1. Data Wrangling and Feature Correction
    - Conversion of Data Types of columns
    - Check and Impute missing (NaN or NULL) values in columns
    - Rename levels of categorical type variables to enhance interpretability
    - Drop irrelevant and redundant columns from raw dataset
2. Exploratory Data Analyses
    - Proportion of 'exits' in raw dataset
    - Tabulation and Visualization of summary statistics of numeric and categorical features
    - Relationship between 'exit' status and numeric and categorical features
    - Linear and Logistic Regression to estimate feature importance in determining 'exit' decision
    - Compare feature importance in Linear and Logistic Regressions
3. Predicting Exit using following Classifier Models (with and without hyperparameter tuning):
    - Logistic Regression
    - Gradient Boosting
    - Random Forest
    - Neural Networks (implemented in Tensorflow)
4. Generate Test Prediction Metrics for each model and compare models

(C) Conclusions:

1. Major Features that contribute to exit decisions:
    - OverTime
    - JobRole (Director of Research, Director of Manufacturing, etc)
    - Business Travel
2. Best model from among the 8 classifier models: Logistic Regression with Tuned Hyperparameters
3. Best Model Prediction Metrics:

- Accuracy: 0.93
- Precision: 0.87
- Recall: 0.61
- F1-Score: 0.71
- ROC-AUC Score: 0.80

4. Best Model Parameters:
   - Penalty: 'l2'
   - C: 0.08685
   - fit_intercept = True
5. Logistic Regression with Hyperparameter Tuning achieves 2% increase in test accuracy over simple Logistic Regression Classifier
6. Logistic Regression with Hyperparameter Tuning achieves 2.5% increase in test accuracy over simple Neural Network Classifier



In [1]:
```
# Importing libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import csv
from csv import reader
import random
import math
import chardet
import warnings
```

```
warnings.filterwarnings('ignore')
import seaborn as sns
import heapq
import re
import plotnine
import copy
import pydot
import graphviz
import datetime
import jinja2
import itertools
from IPython.display import Image
from IPython.core.display import HTML
from graphviz import Source
from scipy.stats import randint as sp_randint
import scipy.stats
from scipy.stats import skew, shapiro, ttest_ind, levene, wilcoxon, ranksums, chi2
from sklearn.linear_model import LinearRegression, LogisticRegression
import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.formula.api import ols, logit
from statsmodels.stats.outliers_influence import variance_inflation_factor
from kmodes.kprototypes import KPrototypes
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from scipy.spatial import distance
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, RandomizedSearchCV, StratifiedKFo
from sklearn.feature_selection import RFECV, RFE
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier, GradientBoo
from sklearn.metrics import make_scorer, accuracy_score, precision_score, recall_score,
from sklearn.tree import export_graphviz
from sklearn.pipeline import Pipeline
from sklearn.datasets import make_classification
from sklearn.inspection import plot_partial_dependence, partial_dependence
from IPython.display import display
from pprint import pprint
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import optimizers
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array, load
from tensorboard.plugins.hparams import api as hp
```

In [2]:

```
## Importing raw data into notebook

# Define paths to raw data, processed data, and graphs to be generated
sourcepath = "D:\Work\Research\Dropbox\Machine Learning\Side Projects\Employee Churn IB
rawdatapath = os.path.join(sourcepath, 'data', 'raw')
finaldatapath = os.path.join(sourcepath, 'data', 'processed')
graphpath = os.path.join(sourcepath, 'reports', 'figures')
rawfilename = 'IBM HR Analytics.csv'
finalfilename = 'IBM_HR_Analytics_Processed.csv'

# Detecting encoding (if any) of the raw data file
rawdata = open(os.path.join(rawdatapath, rawfilename), 'rb').read()
result = chardet.detect(rawdata)
charenc = result['encoding']
```

```
# Load raw data into notebook using the encoding in 'charenc'
IBMData = pd.read_csv(os.path.join(rawdatapath, rawfilename), encoding = charenc)

# Delete any redundant objects
del [rawdata, result, charenc, rawdatapath, rawfilename]
```

(Step 1) Data Wrangling and Feature Correction

- Column Data Type Conversion
- Checks for NULL and/or NaN values and imputate, if required
- Drop Redundant and Irrelevant Columns

First, we visualize the data as a frame.

In [4]:
```
# (0) Visualizing dataframe head

IBMData.head(n=5)
```

Out[4]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationFi
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life Scien
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life Scien
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	Ot
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life Scien
4	27	No	Travel_Rarely	591	Research & Development	2	1	Med

5 rows × 35 columns

Second, we check for the frame for any missing (NaN or NULL) values. If they are present, we take steps to impute the missing values. We also drop two rows: 'StandardHours' and 'Over18' because they contain the same information for all employees. Lastly, we drop the row 'EmployeeNumber' because it contains unique IDs for each employee in our dataframe, and each employee appears only once in the dataset.

In [5]:
```
# (1) Variable names and their data types
# print('Variable Names and Data Types:\n')
# print(IBMData.dtypes)

# (2) Number of observations
#print('--------------------------------------------------------------------------------')
print('\nTotal number of observations in raw dataset: {}'.format(IBMData.shape[0]))

# (3) Check for presence of NULL/NA values in columns
print('--------------------------------------------------------------------------------')
print('\nChecking for NaN values in dataframe columns....\n')
[print(x, IBMData[x].isna().sum()) for x in IBMData.columns if IBMData[x].isna().sum()
if(~IBMData.isna().sum().sum()): print('Raw data has no NaN values.\n')
```

```
print('Checking for Null values in dataframe columns....\n')
[print(IBMData[x].isnull().sum()) for x in IBMData.columns if IBMData[x].isnull().sum()
if(~IBMData.isnull().sum().sum()): print('Raw data has no Null values.\n')
print('--- No imputations required.')

# (4) Drop columns which have no variation across rows
print('-------------------------------------------------------------------------------
print('\nDropping columns which have no variation across rows...\n')
for x in IBMData.columns:
    if(IBMData[x].nunique()==1): # check if the number of unique values == 1
        print('Column dropped: {}'.format(str(x)))
        IBMData.drop([x], axis = 1, inplace = True) # adjust the dataframe inplace by d

# (5) Drop columns which have irrelevant information (like unique IDs)

# We drop the column 'EmployeeNumber' since it contains unique IDs for
# each employee and we do not observe multiple instances of same employee.
print('-------------------------------------------------------------------------------
print('\nDropping Column: "EmployeeNumber" since it has unique ID for each observation.
IBMData.drop(['EmployeeNumber'], axis = 1, inplace = True)
```

```
Total number of observations in raw dataset: 1470
-------------------------------------------------------------------------------
------------------------------

Checking for NaN values in dataframe columns....

Raw data has no NaN values.

Checking for Null values in dataframe columns....

Raw data has no Null values.

--- No imputations required.
-------------------------------------------------------------------------------
------------------------------

Dropping columns which have no variation across rows...

Column dropped: EmployeeCount
Column dropped: Over18
Column dropped: StandardHours
-------------------------------------------------------------------------------
------------------------------

Dropping Column: "EmployeeNumber" since it has unique ID for each observation...
```

Third, we first categorize all columns into one of three types: numeric, categorical, int. 'Numeric' variables include features which are either numeric in type (e.g. MonthlyIncome) or are essentially categorical but have some ordering between the levels (e.g. JobSatisfaction). Categorical features include variables like 'JobRole', 'Department' etc., where the levels cannot be assigned any order. Finally, the exit variable 'Attrition' is typecast as 'Integer' with '1' denoting exit and '0' denoting retention.

In [6]:
```
# (1) Convert all columns into one of three types: float, int, category

# 'Float' type columns include all numeric columns (numvars), as well as some ordinal c
numvars = ['Age', 'DailyRate', 'DistanceFromHome', 'HourlyRate', 'MonthlyIncome', 'Mont
           'PercentSalaryHike', 'TotalWorkingYears', 'TrainingTimesLastYear',
            'YearsAtCompany', 'YearsInCurrentRole', 'YearsSinceLastPromotion', 'YearsWit
ordvars = ['Education', 'EnvironmentSatisfaction', 'JobInvolvement', 'JobLevel', 'JobSa
```

```
          'RelationshipSatisfaction', 'StockOptionLevel', 'WorkLifeBalance']

# 'Category' type columns include all categorical variables with no ordinal levels
catvars = ['BusinessTravel', 'Department', 'EducationField', 'Gender', 'JobRole',
           'PerformanceRating', 'MaritalStatus', 'OverTime']

# 'Integer' type columns include the outcome variable 'Attrition'
intvars = ['Attrition']

# Now convert the columns in numvars and ordvars lists to float
print('Converting some columns to "float"...')
#print(numvars + ordvars)
IBMData[numvars + ordvars] = IBMData[numvars + ordvars].apply(pd.to_numeric, downcast =

# Now convert the columns in catvars to 'category' type
print('------------------------------------------------------------------------------')
print('\nConverting some columns to "category"...')
#print(catvars)
IBMData[catvars] = IBMData[catvars].astype('category');

# Now convert the 'Attrition' column to 'integer' type
print('------------------------------------------------------------------------------')
print('\nConverting "Attrition" column to'' "integer"...')
exit_map = {'Yes':1, 'No':0} # Use a dictionary since this will be used later
IBMData['Attrition'] = IBMData['Attrition'].map(exit_map);


# (2) Rename levels of some categorical variables:
print('------------------------------------------------------------------------------')
print('\nRenaming some levels of some of the categorical variables...')
IBMData['BusinessTravel'].cat.rename_categories({
    'Travel_Rarely':'Rare', 'Travel_Frequently':'Frequent', 'Non-Travel':'No'}, inplace
IBMData['Department'].cat.rename_categories({
    'Research & Development':'R&D','Sales':'Sales','Human Resources':'HR'}, inplace = T
IBMData['EducationField'].cat.rename_categories({
    'Life Sciences':'LifeSc','Technical Degree':'Technical','Human Resources':'HR'}, in
IBMData['JobRole'].cat.rename_categories({
    'Sales Executive':'SalexExec','Research Scientist':'Scientist','Laboratory Technici
    'Manufacturing Director':'DirManufac','Healthcare Representative':'HealthRep','Sale
    'Research Director':'DirResearch','Human Resources':'HR'}, inplace = True)

# (3) Final list of data types and column names
print('------------------------------------------------------------------------------')
maxlength = max(len(numvars + ordvars), len(catvars), 1)
typeframe = pd.DataFrame({'Float':numvars+ordvars,
                          'Category':catvars+['']*(maxlength - len(catvars)),
                          'Integer':['Attrition']+['']*(maxlength - 1)})
print('\nData Types of Feature and Outcome Columns: \n')
print(typeframe)

# Deleting redundant data objects
del [typeframe, maxlength]
```

```
Converting some columns to "float"...
------------------------------------------------------------------------------
------------------------------

Converting some columns to "category"...
------------------------------------------------------------------------------
------------------------------

Converting "Attrition" column to "integer"...
```

```
------------------------------------------------------------------------------------------
------------------------------

Renaming some levels of some of the categorical variables...
------------------------------------------------------------------------------------------
------------------------------

Data Types of Feature and Outcome Columns:

                      Float           Category    Integer
0                       Age     BusinessTravel  Attrition
1                 DailyRate         Department
2           DistanceFromHome     EducationField
3                 HourlyRate             Gender
4              MonthlyIncome            JobRole
5                MonthlyRate  PerformanceRating
6         NumCompaniesWorked      MaritalStatus
7          PercentSalaryHike           OverTime
8          TotalWorkingYears
9      TrainingTimesLastYear
10            YearsAtCompany
11        YearsInCurrentRole
12   YearsSinceLastPromotion
13      YearsWithCurrManager
14                 Education
15   EnvironmentSatisfaction
16            JobInvolvement
17                  JobLevel
18           JobSatisfaction
19  RelationshipSatisfaction
20          StockOptionLevel
21           WorkLifeBalance
```

(Step 2) Exploratory Data Analyses

(Step 2.a) Exploring Outcome Variable ('Attrition')

- Proportion of workers leaving

(Step 2.b) Exploring Feature Variables

- Summary statistics of numeric variables (Table and Visualizations)
- Frequency Distributions of categorical variables (Table and Visualizations)
- Pairwise scatter plots of numeric variables (with and without categorical types)
- Summary statistics of numeric variables grouped by categorical types
- Cross-category frequency visualizations

(Step 2.c) Exploring Relationship between Outcome and Feature Variables

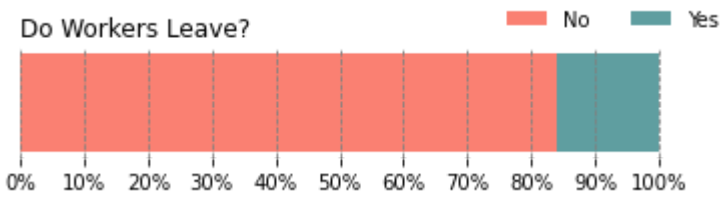
- Statistical Tests of difference in each numeric variable between exit decisions
- Statistical Tests of difference in proportion of exits between any two levels of each categorical variable
- Linear Regression (Linear Probability Model) to explain joint contributions of all numeric and categorical features to exit decisions
- Logistic Regression to explain joint contributions of all numeric and categorical features to exit decisions
- Comparison of feature contributions in Linear and Logistic Regressions

(Step 2.a) Proportion of employees who leave

In the first step, we compute the raw proportion of exits in the data, and show this graphically in a bar chart.

In [7]:
```
## (A) Exploring Outcome Variable: Attrition

print('Percentage of employees who leave: {:2.3%}\n'.format(IBMData['Attrition'].sum()/

# Show graphically the fraction of workers leaving
df_grouped = IBMData['Attrition'].value_counts()/len(IBMData)
fracLeave = pd.DataFrame({df_grouped.index[0]:[df_grouped[0]],
                          df_grouped.index[1]:[df_grouped[1]]}) # Create a dataframe wi
fracLeave.columns = fracLeave.columns.map(dict(map(reversed, exit_map.items()))) # Chan
fields = fracLeave.columns.to_list() # Get a list of the column names
colors = ['salmon', 'cadetblue']
left = len(fracLeave)*[0]

fig, ax = plt.subplots(figsize = (6,1))
for idx, name in enumerate(fields):
    plt.barh(fracLeave.index, fracLeave[name], left = left, color = colors[idx])
    left = left + fracLeave[name]
plt.title('Do Workers Leave?', loc = 'left')
plt.legend(fracLeave.columns.to_list(), ncol = 2, frameon = False, bbox_to_anchor = ([0
ax.spines['right'].set_visible(False)
ax.spines['left'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.spines['bottom'].set_visible(False)
xticks = np.arange(0,1.1,0.1)
xlabels = ['{}%'.format(i) for i in np.arange(0,101,10)]
plt.xticks(xticks, xlabels)
ax.get_yaxis().set_ticks([])
ax.xaxis.grid(color='gray', linestyle='dashed')
plt.show()

# Delete redundant data objects
del [df_grouped, fracLeave, fields, colors, left, idx, name, fig, ax]
```

```
Percentage of employees who leave: 16.122%
```



The plot above shows that over 80% of employees do not leave the firm. Only about 16% of employee exit.

(Step 2.b) Summary Statistics of Numeric and Categorical Feature Variables

In the second step, we compute, tabulate, and visualize summary statistics of numeric and categorical features. Here, we conduct the following types of analyses:

- (A) Tabulation of Summary Statistics of Numeric Features
- (B) Visualization (Histogram + Boxplot) of Select Numeric Features

- (C) Visualization (Category Plots) of Frequency Distribution of the levels of each categorical feature
- (D) Visualization (Heatmap) of Pairwise Correlation between numeric features
- (E) Tabulation of Summary Statistics of Numeric Features Grouped by Select Categorical Features
- (F) Visualization (Histogram + Boxplot) of Select Numeric Features Grouped by Select Categorical Features
- (G) Visualization (Category Plots) of Frequency Distributions of Select categorical features by other categorical features

(A) Summary Statistics of Numeric Features

In [8]:
```
# (1) Summary Statistics of Numeric Variables

print('Descriptive Summary Statistics of all Numeric Variables:\n')
floatvar_stats = IBMData[numvars + ordvars].describe().transpose()
floatvar_stats
```

```
Descriptive Summary Statistics of all Numeric Variables:
```

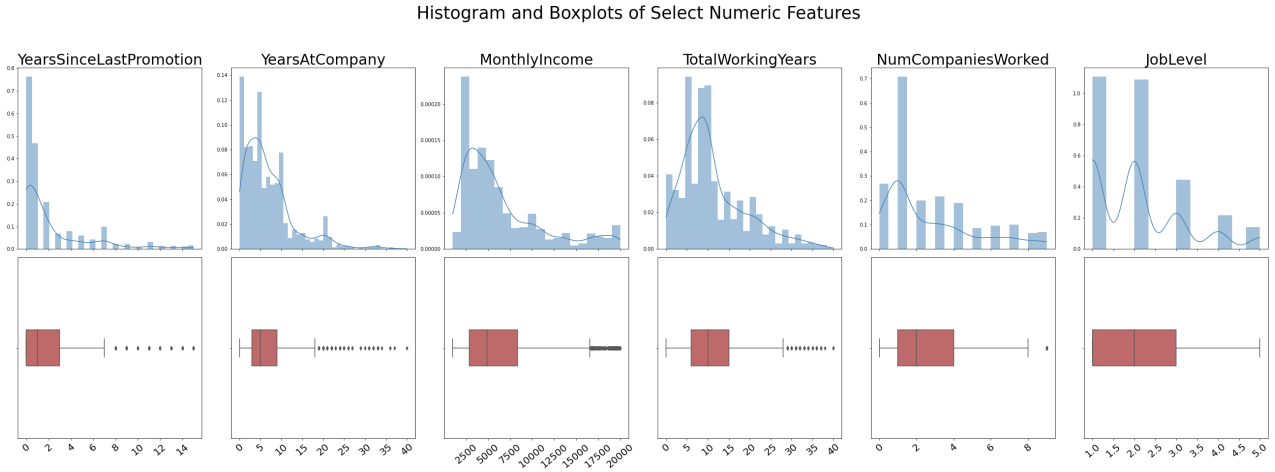
Out[8]:

	count	mean	std	min	25%	50%	75%	max
Age	1470.0	36.923809	9.135373	18.0	30.0	36.0	43.00	60.0
DailyRate	1470.0	802.485718	403.509094	102.0	465.0	802.0	1157.00	1499.0
DistanceFromHome	1470.0	9.192517	8.106865	1.0	2.0	7.0	14.00	29.0
HourlyRate	1470.0	65.891159	20.329428	30.0	48.0	66.0	83.75	100.0
MonthlyIncome	1470.0	6502.931152	4707.956543	1009.0	2911.0	4919.0	8379.00	19999.0
MonthlyRate	1470.0	14313.103516	7117.786133	2094.0	8047.0	14235.5	20461.50	26999.0
NumCompaniesWorked	1470.0	2.693197	2.498009	0.0	1.0	2.0	4.00	9.0
PercentSalaryHike	1470.0	15.209524	3.659938	11.0	12.0	14.0	18.00	25.0
TotalWorkingYears	1470.0	11.279592	7.780782	0.0	6.0	10.0	15.00	40.0
TrainingTimesLastYear	1470.0	2.799320	1.289271	0.0	2.0	3.0	3.00	6.0
YearsAtCompany	1470.0	7.008163	6.126525	0.0	3.0	5.0	9.00	40.0
YearsInCurrentRole	1470.0	4.229252	3.623137	0.0	2.0	3.0	7.00	18.0
YearsSinceLastPromotion	1470.0	2.187755	3.222430	0.0	0.0	1.0	3.00	15.0
YearsWithCurrManager	1470.0	4.123129	3.568136	0.0	2.0	3.0	7.00	17.0
Education	1470.0	2.912925	1.024165	1.0	2.0	3.0	4.00	5.0
EnvironmentSatisfaction	1470.0	2.721769	1.093082	1.0	2.0	3.0	4.00	4.0
JobInvolvement	1470.0	2.729932	0.711561	1.0	2.0	3.0	3.00	4.0
JobLevel	1470.0	2.063946	1.106940	1.0	1.0	2.0	3.00	5.0
JobSatisfaction	1470.0	2.728571	1.102846	1.0	2.0	3.0	4.00	4.0
RelationshipSatisfaction	1470.0	2.712245	1.081209	1.0	2.0	3.0	4.00	4.0

	count	mean	std	min	25%	50%	75%	max
StockOptionLevel	1470.0	0.793878	0.852077	0.0	0.0	1.0	1.00	3.0
WorkLifeBalance	1470.0	2.761225	0.706476	1.0	2.0	3.0	3.00	4.0

(B) (Histogram + Boxplot) of Select Numeric Features

In [9]:
```
# Visualization of Summary Statistics of Select Numeric Variables
# Select 6 numeric variables with the highest value of skewness (absolute value)

# Compute the skewness values for the distribution of each numeric variable
skew_vals = skew(IBMData[numvars+ordvars], axis = 0, bias = False)
# Find the index of largest 6 absolute skewness values
indexvals = heapq.nlargest(6, range(len(skew_vals)), key = lambda x: abs(skew_vals[x]))
# Now select the numeric variables with largest absolute skewness values
select_floatvars = [(numvars + ordvars)[x] for x in indexvals]
# select_floatvars = ['Age', 'DistanceFromHome', 'MonthlyIncome', 'PercentSalaryHike',

plt.rcParams['patch.linewidth'] = 0
plt.rcParams['patch.edgecolor'] = 'none'
fig, ax = plt.subplots(2, len(select_floatvars), figsize = (6*len(select_floatvars), 6*
for i, item in enumerate(select_floatvars):

    # Plot the histogram first with kernel density estimates
    plt.subplot(2, len(select_floatvars), i+1)
    g = sns.histplot(IBMData, x = select_floatvars[i], stat = 'density', color = 'steel
    g.set_title(select_floatvars[i], fontsize = 30)
    g.set_ylabel('')
    g.set_xlabel('')
    g.set_xticklabels([])

    # Then plot the boxplot with median values
    plt.subplot(2, len(select_floatvars), i+7)
    f = sns.boxplot(IBMData[item], color = 'indianred', width = 0.2, orient = 'v')
    plt.draw()
    labels = [newitem.get_text() for newitem in f.get_xticklabels()]
    f.set_xlabel('')
    f.set_xticklabels(labels, fontsize = 20, rotation = 40)
plt.tight_layout()
plt.suptitle('Histogram and Boxplots of Select Numeric Features', fontsize = 35, y = 1.
plt.show()
```

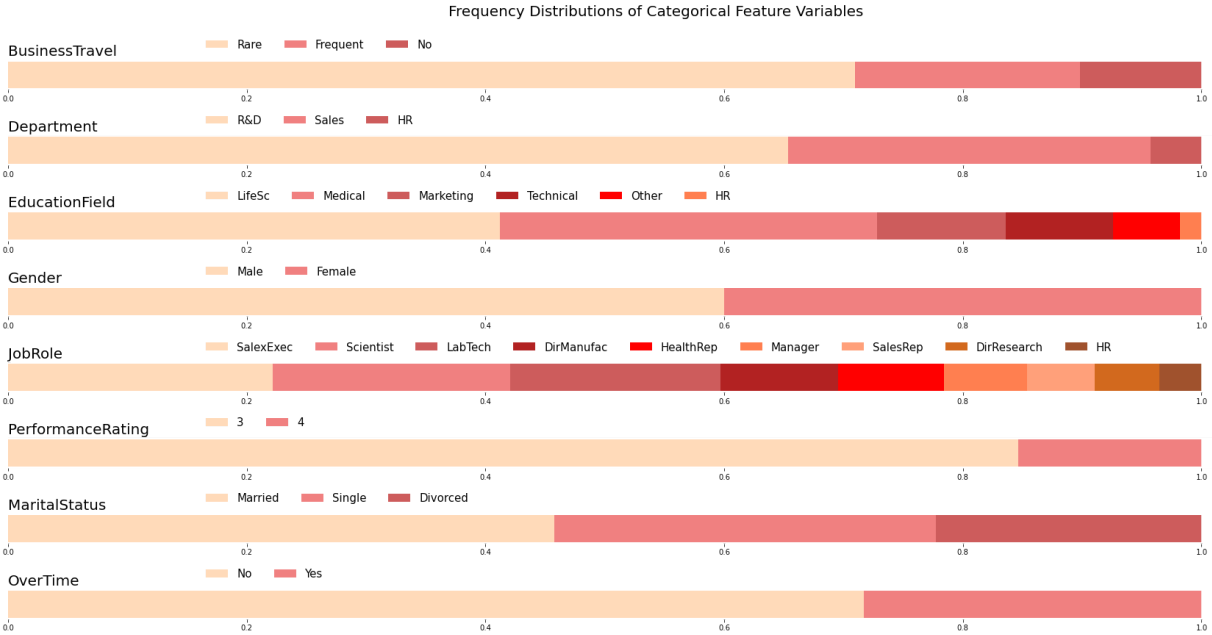


As the histogram plots above suggest, the numeric features plotted above clearly do not follow a

normal distribution, but are mostly right skewed.

(C) Category Plots of Frequency Distributions of the levels of each categorical feature

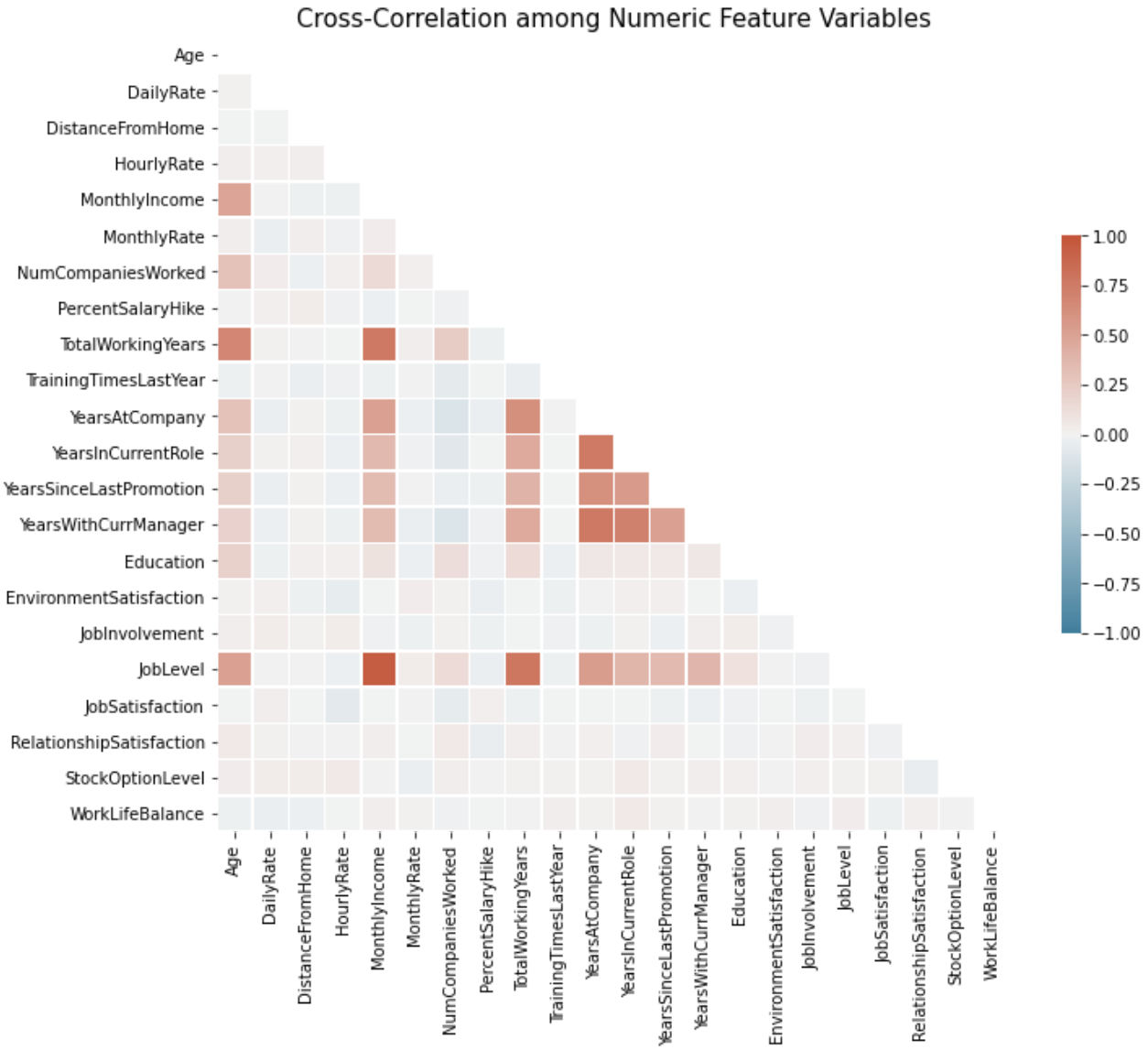
In [10]:
```
## Frequency Distributions of Select Categorical Variables

# Define a function which takes in the variable and then plots a frequency bar
def plot_catfreq(data, varname, axes, colors):

    # First create a dataframe which contains the frequency of the levels in the column
    df_grouped = pd.DataFrame(data[varname].value_counts()/len(data)).transpose()
    fields = df_grouped.columns.to_list()
    left = len(df_grouped)*[0]
    for idx, name in enumerate(fields):
        axes.barh(df_grouped.index, df_grouped[name], left = left, color = colors[idx],
        left = left + df_grouped[name]
    axes.spines['right'].set_visible(False)
    axes.spines['left'].set_visible(False)
    axes.spines['top'].set_visible(False)
    axes.spines['bottom'].set_visible(False)
    axes.set_title(varname, loc = 'left', fontsize = 20)
    axes.legend(fields, ncol = data[varname].nunique(), frameon = False, bbox_to_anchor
    axes.set_yticks([])
    xticks = np.arange(0,1.1,0.1)
    xlabels = ['{}%'.format(i) for i in np.arange(0,101,10)]

# Now define the parameters to pass to the function
colors = ['peachpuff', 'lightcoral','indianred','firebrick','red','coral','lightsalmon'
fig, ax = plt.subplots(nrows = len(catvars), ncols = 1, figsize = (25, 1.5*len(catvars)
for i, axes in enumerate(ax.flatten()):
    plot_catfreq(IBMData, catvars[i], axes, colors)
plt.suptitle('Frequency Distributions of Categorical Feature Variables', fontsize = 20,
plt.tight_layout()
plt.show()
```



In the graph above, each row bar represents a categorical feature. The division of each bar into levels represent the proportion (or fraction) of each level for that categorical feature. For example, the second bar on 'Department' implies that roughly 65% of the employees work in the 'R&D' Department, 30% work in the 'Sales' Department, and the rest work in the 'HR' Department.

(D) Heatmap of pairwise correlation between numeric features

In [11]:
```
# Cross Correlations of Numeric Variables

# First compute the correlation matrix
corr = IBMData[numvars + ordvars].corr()
# Mask the upper triangle
mask = np.triu(np.ones_like(corr, dtype = bool))
fig, ax = plt.subplots(figsize = (11,9))
cmap = sns.diverging_palette(230, 20, as_cmap = True)
g = sns.heatmap(corr, mask = mask, cmap = cmap, vmax = 1.0, vmin = -1.0, center = 0, sq
g.set_title('Cross-Correlation among Numeric Feature Variables', fontsize = 15);
```



The heatmap of cross-correlations suggest that most numeric features are not very correlated with other numeric features. There are however, some natural correlations between numeric variables - for example, between TotalWorkingYears and MonthlyIncome, JobLevel and MonthlyIncome, YearsWithCurrManager and YearsAtCompany. The higher values of positive correlation are represented by boxes that are increasingly red in color.

(E) Summary Statistics of Numeric Features Grouped By Select Categorical Features

In [12]:
```
# Summary Statistics of numeric variables grouped by categorical variables
# Here we show the mean, median, and standard deviations of each numeric variable group
```

```
# Create a dataframe in which the rows are the variables, there are columns for each st

grouping_var = 'Department'
stats = IBMData[numvars + ordvars + [grouping_var]].groupby('Department').describe()
statlist = ['mean', '50%', 'std']
statlist_changed = ['Mean', 'Median', 'Standard Deviation']
maxlevels = IBMData[grouping_var].nunique()

for counter, item in enumerate(numvars+ordvars):
    newdf = pd.DataFrame()
    for stat in statlist:
        newdf = pd.concat([newdf,
            stats.loc[:,item][stat].reset_index().pivot_table(index = [], columns = gro
            axis = 1, ignore_index = False)
    headers = list(itertools.chain.from_iterable(itertools.repeat(x, maxlevels) for x i
    newdf.columns = pd.MultiIndex.from_tuples(list(zip(headers, newdf.columns)))
    newdf.insert(0, 'Variable', item)
    if(counter == 0):
            statdf = newdf.copy(deep = True)
    else:
        statdf = statdf.append(newdf, ignore_index = True)
pd.set_option('display.expand_frame_repr', False)
print('Summary Statistics of Numeric Variables by Department')
statdf
```

```
Summary Statistics of Numeric Variables by Department
```

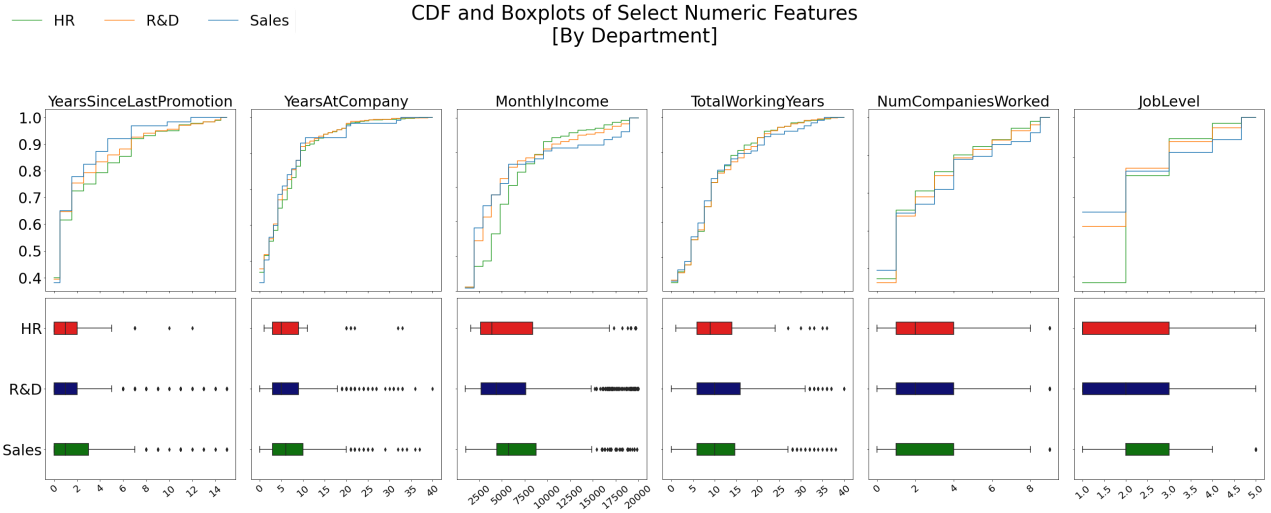
Out[12]:

	Variable	Mean			Median			
		HR	R&D	Sales	HR	R&D	Sales	
0	Age	37.809525	37.042664	36.542603	37.0	36.0	35.0	9.2
1	DailyRate	751.539673	806.851196	800.275757	788.0	810.0	770.5	426.2
2	DistanceFromHome	8.698413	9.144641	9.365471	6.0	7.0	7.0	8.1
3	HourlyRate	64.301590	66.167534	65.520180	59.0	66.0	66.0	21.5
4	MonthlyIncome	6654.507812	6281.252930	6959.172852	3886.0	4374.0	5754.5	5788.7
5	MonthlyRate	13492.984375	14284.866211	14489.793945	12832.0	14242.0	14419.5	7426.8
6	NumCompaniesWorked	2.936508	2.733611	2.571749	2.0	2.0	1.0	2.8
7	PercentSalaryHike	14.761905	15.291363	15.096413	14.0	14.0	14.0	3.6
8	TotalWorkingYears	11.555555	11.342352	11.105381	9.0	10.0	10.0	8.8
9	TrainingTimesLastYear	2.555556	2.792924	2.847534	2.0	3.0	3.0	1.2
10	YearsAtCompany	7.238095	6.864724	7.284753	5.0	5.0	6.0	6.8
11	YearsInCurrentRole	3.539683	4.155047	4.486547	2.0	3.0	3.0	2.8
12	YearsSinceLastPromotion	1.777778	2.137357	2.354260	1.0	1.0	1.0	2.5
13	YearsWithCurrManager	3.666667	4.084287	4.271300	3.0	3.0	3.0	2.9
14	Education	2.968254	2.899064	2.934978	3.0	3.0	3.0	0.9
15	EnvironmentSatisfaction	2.682540	2.744017	2.679372	3.0	3.0	3.0	1.0
16	JobInvolvement	2.746032	2.741935	2.701794	3.0	3.0	3.0	0.7

	Variable	Mean			Median			
		HR	R&D	Sales	HR	R&D	Sales	
17	JobLevel	2.031746	1.977107	2.255605	1.0	2.0	2.0	1.3
18	JobSatisfaction	2.603175	2.726327	2.751121	3.0	3.0	3.0	1.0
19	RelationshipSatisfaction	2.888889	2.708637	2.695067	3.0	3.0	3.0	1.0
20	StockOptionLevel	0.777778	0.804370	0.773543	1.0	1.0	1.0	0.8
21	WorkLifeBalance	2.920635	2.725286	2.816144	3.0	3.0	3.0	0.7

(F) (Historgram + Boxplot) of Select Numeric Features Grouped by Select Categorical Features

In [13]:
```
# Now plot histogram and boxplots of select numeric variables against department

plt.rcParams['patch.linewidth'] = 0
plt.rcParams['patch.edgecolor'] = 'none'
fig, ax = plt.subplots(2, len(select_floatvars), figsize = (6*len(select_floatvars), 6*
newlist = []
for i, item in enumerate(select_floatvars):

    # Plot the histogram first with kernel density estimates
    plt.subplot(2, len(select_floatvars), i+1)
    g = sns.histplot(IBMData, x = select_floatvars[i], hue = grouping_var, stat = 'dens
                     color = ['red','navy','green'], cumulative = True, common_norm = F
    plt.draw()
    g.set_title(select_floatvars[i], fontsize = 30)
    g.set_ylabel('')
    g.set_xlabel('')
    g.set_xticklabels([])
    if(i == 0):
        labels = [newitem.get_text() for newitem in g.get_yticklabels()]
        g.set_yticklabels(labels, fontsize = 30)
    else:
        g.set_yticklabels([])
    newlist.append(g)

    # Then plot the boxplot with median values
    plt.subplot(2, len(select_floatvars), i+7)
    f = sns.boxplot(data = IBMData, x = item, y = grouping_var, width = 0.2, palette =
    plt.draw()
    labels = [newitem.get_text() for newitem in f.get_xticklabels()]
    f.set_xlabel('')
    f.set_ylabel('')
    f.set_xticklabels(labels, fontsize = 20, rotation = 40)
    if(i == 0):
        labels = [newitem.get_text() for newitem in f.get_yticklabels()]
        f.set_yticklabels(labels, fontsize = 30)
    else:
        f.set_yticklabels([])

fig.legend(newlist, labels = ['HR','R&D','Sales'], loc = 'upper left', ncol = maxlevels
plt.tight_layout()
plt.suptitle('CDF and Boxplots of Select Numeric Features\n[By Department]', fontsize =
plt.show()
```

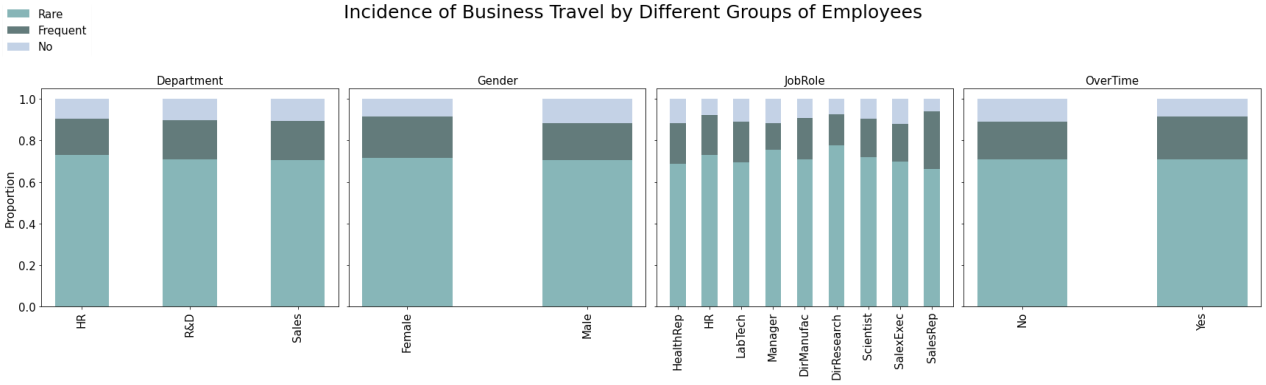


The cumulative distribution plots and boxplots above suggest that there is little difference between the summary statistics of these numeric features by 'Department'. The only numeric feature above which seems to differ between 'Departments' is JobLevel. Here, the JobLevel is on an average higher in the 'HR' Department than in 'R&D' and 'Sales'.

(G) Category Plots of Select Categorical Features by Another Categorical Feature

In [15]:

```
# Plot one or more categorical variables by another categorical variable:
# Categorize business travel by department, gender, jobrole, and overtime

select_catvars = ['BusinessTravel', 'Department', 'Gender', 'JobRole', 'OverTime']
maxlevels = max([IBMData[item].nunique() for item in select_catvars[1:]])

leglabels = []
fig, ax = plt.subplots(1, len(select_catvars)-1, figsize = (6*4, 6))
for counter, item in enumerate(select_catvars[1:]):

    newdf = pd.crosstab(IBMData[item], IBMData[select_catvars[0]], margins = True)
    newdf = newdf.iloc[:,:-1].div(newdf.iloc[:,-1], axis = 0).iloc[:-1,:].reset_index()
    newdf = pd.melt(newdf, id_vars = item, var_name = select_catvars[0], value_name = '

    plt.subplot(1, len(select_catvars)-1, counter+1)
    g = sns.histplot(newdf, x = item, hue = select_catvars[0], weights = 'Proportion',
                     shrink = 0.5, legend = False, palette = ['lightsteelblue','darksla
    plt.draw()
    g.set_title(item, fontsize = 15)
    g.set_xlabel('')
    labels = [newitem.get_text() for newitem in g.get_xticklabels()]
    g.set_xticklabels(labels, fontsize = 15, rotation = 90)
    if(counter == 0):
        labels = [newitem.get_text() for newitem in g.get_yticklabels()]
        g.set_yticklabels(labels, fontsize = 15)
        g.set_ylabel('Proportion', fontsize = 15)
    else:
        g.set_yticklabels([])
        g.set_ylabel('')
    leglabels.append(g)
fig.legend(leglabels, labels = ['Rare','Frequent','No'], loc = 'upper left', fontsize =
plt.tight_layout()
plt.suptitle('Incidence of Business Travel by Different Groups of Employees', fontsize
```



(Step 2.c) Relationship Between Exit Decision and Feature Variables

In this section we evaluate the relation between the outcome variable (exit decision) and numeric and categorical features. More specifically, we conduct the following types of analyses:

- (A) Statistical Test of Difference between Exit Status for each numeric feature
- (B) Statistical Test of Difference between Proportion of Exits for any two levels of each categorical feature
- (C) Linear Regression/Linear Probability Model (LPM) to estimate feature importance in determining exit decision
- (D) Logistic Regression and Marginal Effects Computation to estimate feature importance in determining exit decision
- (E) Comparison of Feature Importance in Explaining Exit Decision between Linear and Logistic Regression Models

(A) Statistical Test for Difference Between Exit Status for each Numeric Feature

We first test whether the distribution of a particular numeric feature is statistically different between exit decisions. For example, we check whether the age (a numeric feature) distribution for those who exit is statistically different from the age distribution for those who decide to stay. We conduct this analyses separately for each numeric feature.

To do this we follow the steps below:

- Use Shapiro-Wilk test to check whether the distribution of the numeric feature is separately normal for those who exit and those who stay.
- Use independent t-test to check for statistical difference in the means of the numeric feature, if normality is satisfied for both those who stay and those who exit.
- If normality fails for any one of the groups then do the following:
    - Use two-sample Wilcoxon Test (also known as two-sample Signed Rank Test) to check non-parameterically the statistical difference between the distributions of the numeric feature for the two types of exit decisions.

To decide on a threshold for statistical significance, we choose a significance level or 5% (or alternatively, a confidence level of 95%) for each type of statistical test. Therefore p-value < 0.05 for each test, should indicate statistical significance.

First, we check which of the numeric features follow normal distribution for both types of exit decisions. The results are shown below.

In [16]:
```
# Create two lists, one to store the normal numeric variables, and the other for non-no
normal_floatvars = []
nonnormal_floatvars = []
confidence_level = 0.05

for counter, item in enumerate(numvars+ordvars):
    if(shapiro(IBMData[IBMData['Attrition']==1].Age).pvalue > confidence_level and
      shapiro(IBMData[IBMData['Attrition']==0].Age).pvalue > confidence_level):
        normal_floatvars.append(item)
    else:
        nonnormal_floatvars.append(item)

print('Numeric Features which follow normal distribution, conditional on both exit stat
print(normal_floatvars)
print('\nNumeric Features which do not follow normal distribution, conditional on at le
print(nonnormal_floatvars)

if(len(normal_floatvars)==0):
    print('\n\nTherefore none of the numeric features follow normal distribution, condi
elif(len(nonnormal_floatvars)==0):
    print('\n\nTherefore none of the numeric features deviate from the normal distribut
```

```
Numeric Features which follow normal distribution, conditional on both exit status:
[]

Numeric Features which do not follow normal distribution, conditional on at least one ex
it status:
['Age', 'DailyRate', 'DistanceFromHome', 'HourlyRate', 'MonthlyIncome', 'MonthlyRate',
'NumCompaniesWorked', 'PercentSalaryHike', 'TotalWorkingYears', 'TrainingTimesLastYear',
'YearsAtCompany', 'YearsInCurrentRole', 'YearsSinceLastPromotion', 'YearsWithCurrManage
r', 'Education', 'EnvironmentSatisfaction', 'JobInvolvement', 'JobLevel', 'JobSatisfacti
on', 'RelationshipSatisfaction', 'StockOptionLevel', 'WorkLifeBalance']


Therefore none of the numeric features follow normal distribution, conditional on both e
xit status.
```

Since no numeric feature follows normal distribution for both types of exit decisions, we will apply the two-sample Wilcoxon Test to check for the statistical difference between the distributions of the numeric feature by exit decision.

In [17]:
```
# Since no numeric feature follows normal distribution we will apply the one-sample Wil

wilcoxontest_df = pd.DataFrame()
pval_list = []

print('Conducting 2-sample Wilcoxon Signed Rank Test to check for statistical differenc
print('\nNull Hypothesis: The two distributions by exit status are not statistically di
for item in nonnormal_floatvars:
    pval_list.append(ranksums(IBMData[IBMData['Attrition']==1][item], IBMData[IBMData['

wilcoxontest_df['Numeric Feature'] = nonnormal_floatvars
wilcoxontest_df['pValue Wilcoxon Test'] = pval_list

def highlightcells(s):
    if(s['pValue Wilcoxon Test'] < confidence_level):
        return ['background-color: tomato']*wilcoxontest_df.shape[1]
```

```
    else:
        return ['background-color: white']*wilcoxontest_df.shape[1]

print('\nTable below highlights numeric features for which distributions are statistica
wilcoxontest_df.style.apply(highlightcells, axis = 1)
```

```
Conducting 2-sample Wilcoxon Signed Rank Test to check for statistical difference betwee
n distributions of numeric features by exit status...

Null Hypothesis: The two distributions by exit status are not statistically different.


Table below highlights numeric features for which distributions are statistically differ
ent by exit status:
```

Out[17]:

	Numeric Feature	pValue Wilcoxon Test
0	Age	0.000000
1	DailyRate	0.028999
2	DistanceFromHome	0.002473
3	HourlyRate	0.797588
4	MonthlyIncome	0.000000
5	MonthlyRate	0.558692
6	NumCompaniesWorked	0.254566
7	PercentSalaryHike	0.368351
8	TotalWorkingYears	0.000000
9	TrainingTimesLastYear	0.058464
10	YearsAtCompany	0.000000
11	YearsInCurrentRole	0.000000
12	YearsSinceLastPromotion	0.049885
13	YearsWithCurrManager	0.000000
14	Education	0.266529
15	EnvironmentSatisfaction	0.000367
16	JobInvolvement	0.000054
17	JobLevel	0.000000
18	JobSatisfaction	0.000144
19	RelationshipSatisfaction	0.115046
20	StockOptionLevel	0.000000
21	WorkLifeBalance	0.082243

The table above shows that variables like Age, DailyRate, DistanceFromHome, MonthlyIncome, TotalWorkingYears, YearsAtCompany, YearsInCurrentRole, YearsSinceLastPromotion, YearsWithCurrManager, EnvironmentSatisfaction, JobInvolvement, JobLevel, JobSatisfaction, StockOptionLevel, are statistically different for those who exit and those who stay. In other words, these features are possibly important determinants of the exit decision.

Some of the features mentioned above make sense. For example young people are probably more likely to switch jobs than older people. If an employee lives far away from where they work, they are more likely to want to switch into jobs closer to home. MonthlyIncome might incentivize employees to switch jobs into those with better pay. However, in the same breath, it makes little sense that MonthlyRate or PercentSalaryHike should not have any significant effect on the exit decision. Therefore, the results shown above are only suggestive and should not be taken as perfect determinants of exit decision.

(B) Statistical Test of Difference in Proportion of Exits between any Two Levels of each Categorical Feature

We now test for the statistical significance of the difference in the proportion of exits between any two levels of a cateogorical variable, and we do this or each categorical variable separately. This is implemented using the Marascuillo procedure. Below, we show only those categorical features, for which I detect significant difference between proportions of exits between at least one combination of levels for that categorical variable. The categorical features not shown below suggest, that proportion of exits is not different between any two levels of that variable.

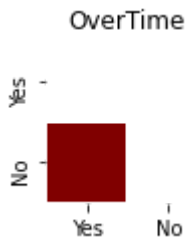
In [18]:
```
print('"Red" indicates the combination of levels for which there is a statistical diffe


# Define a function which computes the significance level of the proportion between two

def compute_significance_propdiff(x, y, alpha, dof):

    #print(len(x), len(y), sum(x), sum(y))
    zval = np.sqrt(chi2.ppf(1.-alpha/2, dof))*np.sqrt(sum(x)*(len(x)-sum(x))/len(x)**3
    if(abs(sum(x)/len(x) - sum(y)/len(y)) < zval):
        return 0.
    else:
        return 1.

# Arrange the catvars list in ascending order of the number of levels
numlevels = [IBMData[item].nunique() for item in catvars]
catvars = [x for (index, x) in sorted(zip(numlevels, catvars))]

for item in catvars:

    # Define a dataframe to store the significance outcomes
    numlevels = IBMData[item].nunique()
    newframe = np.zeros((numlevels, numlevels))

    # Iterate over each level of the variable
    levels = IBMData[item].unique().to_list()
    for c1, l1 in enumerate(levels):
        for c2, l2 in enumerate(levels):
            if(c1 > c2):
                newframe[c1, c2] = compute_significance_propdiff(IBMData[IBMData[item]=
                                                                 IBMData[IBMData[item]=
                                                                 confidence_level, numl
    # Plot if at least one comparison yields significane differences in the proportions
    if(np.any(newframe > 0.5)):
        print('\nFeature: "'+item+'" ----- Statistical Difference Detected in proportio
        newframe = pd.DataFrame(newframe)
        newframe.columns = levels
        newframe.index = levels
```

```
        #print(newframe)

        mask = newframe < 0.5
        fig, ax = plt.subplots(figsize = (0.75*numlevels,0.75*numlevels))
        g = sns.heatmap(newframe, mask = mask, cmap = ['maroon','antiquewhite'],
                        square = True, linewidth = 0.5, cbar = False)
        g.set_title(item)
        plt.show()
    else:
        print('\nFeature: "'+item+'" ----- No Statistical Difference in proportion of e
```

"Red" indicates the combination of levels for which there is a statistical difference in proportion of exits:

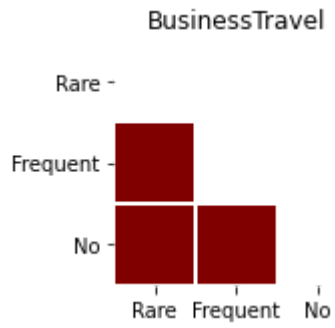
Feature: "Gender" ----- No Statistical Difference in proportion of exits between levels.

Feature: "OverTime" ----- Statistical Difference Detected in proportion of exits between some levels.

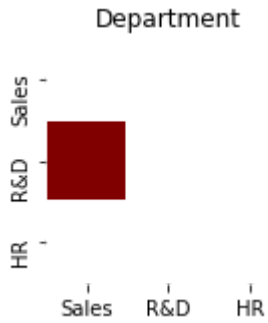


Feature: "PerformanceRating" ----- No Statistical Difference in proportion of exits betw
een levels.

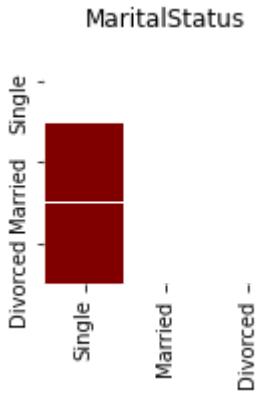
Feature: "BusinessTravel" ----- Statistical Difference Detected in proportion of exits b
etween some levels.



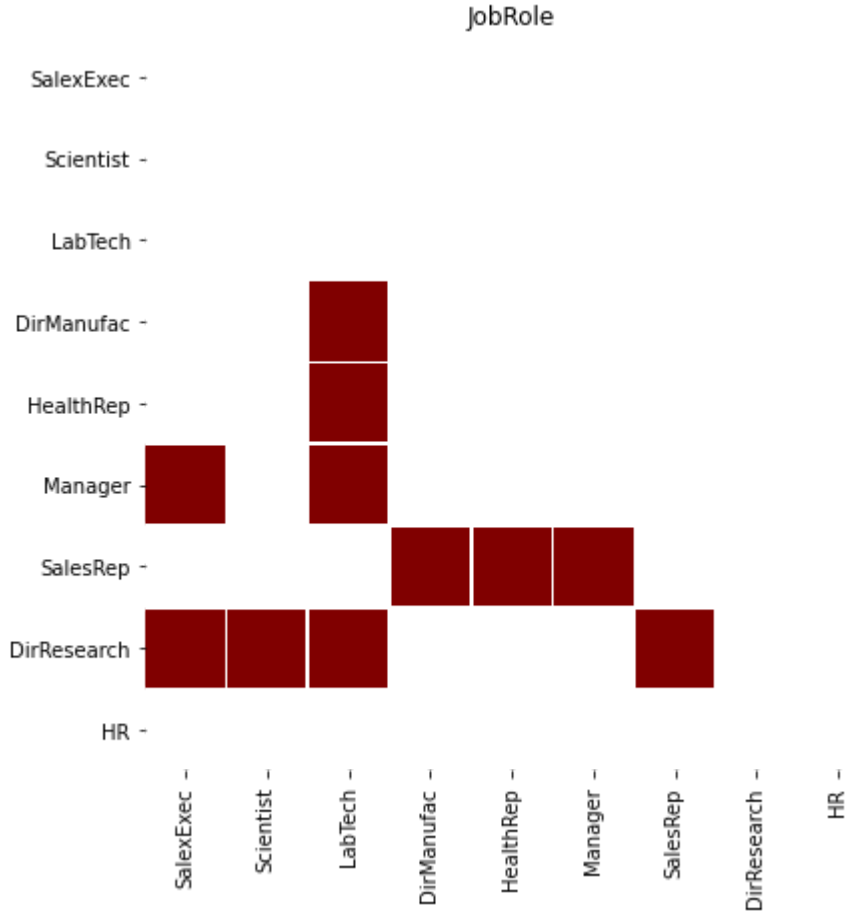
Feature: "Department" ----- Statistical Difference Detected in proportion of exits betwe
en some levels.



Feature: "MaritalStatus" ----- Statistical Difference Detected in proportion of exits be
tween some levels.




```
Feature: "EducationField" ----- No Statistical Difference in proportion of exits between
levels.

Feature: "JobRole" ----- Statistical Difference Detected in proportion of exits between
some levels.
```



The analyses above shows that proportions of exits are statistically different at least between some levels for the categorical features: JobRole, MaritalStatus, Department, BusinessTravel, and OverTime. There is no statistical difference in the proportion of exits between any levels of other categorical variables.

In 'JobRole' for example, proportion of exits are statistically different for those in HealthRep and LabTech positions, while there is no statistical difference in the proportion of exits between HealthRep and Scientist positions.

(C) Linear Regression/Linear Probability Model (LPM) to estimate feature importance in determining exit decision

In this analysis, we regress the binary 'Attrition' variable on a linear function of all features and then estimate the coefficients for each feature. For each categorical variable we drop one level, which we call the 'base level'. The coefficients on each feature (or each level of categorical features) can be interpreted as the marginal effect of that feature (or level) on the employee's exit decision. We conduct a similar analyses with Logistic Regression and then compare results with the Linear Regression case.

To ensure that we can compare coefficients for different features, we scale each numeric feature by subtracting from each numeric feature its mean and then dividing it by the standard deviation of the numeric feature.

In [19]:
```
scaler = StandardScaler()

# Scale only the numeric variables and retain the categorical variables as such
IBMData_numscaled = pd.concat([pd.DataFrame(scaler.fit(IBMData[numvars + ordvars]).tran
                                            columns = IBMData[numvars + ordvars].columns
                               IBMData[catvars + ['Attrition']]], axis = 1)
print('Shape of old dataframe: {}'.format(IBMData.shape))
print('Shape of scaled dataframe: {}'.format(IBMData_numscaled.shape))
```

```
Shape of old dataframe: (1470, 31)
Shape of scaled dataframe: (1470, 31)
```

In [20]:
```
# Define the base-levels to be dropped for each categorical feature as a dictionary
# Choice of base-level is generally not arbitrary. We drop the level which we believe w
baselevel_dict = {'Gender':'"Female"',
                  'OverTime':'"No"',
                  'PerformanceRating':3,
                  'BusinessTravel':'"No"',
                  'Department':'"HR"',
                  'MaritalStatus':'"Divorced"',
                  'EducationField':'"Other"',
                  'JobRole':'"HR"'}

# Now generate the specification for LPM as a string
formula = 'Attrition ~ '
for item in (numvars + ordvars):
    formula = formula + str(item) + ' + '
for counter, item in enumerate(catvars):
    if(counter == len(catvars)-1):
        formula = formula + 'C(' + str(item) + ', Treatment(reference = ' + str(baselev
    else:
        formula = formula + 'C(' + str(item) + ', Treatment(reference = ' + str(baselev

# Now fit the LPM to the scaled data
LPM_Inference = smf.ols(data = IBMData_numscaled, formula = formula).fit()
```

Now we arrange these coefficients by ascending order of their absolute values, to find out which features have the highest contribution to exit decisions. We also drop the intercept in this plot.

In [21]:
```
# Define a function that can take the transformed feature name and then convert it into

# Define a function which takes in a dataframe with feature names, actual effects, and
# and sorts the dataframe by absolute value of effects and corrects the feature names.

def modify_frame(oldframe, effect_var, name_var):
```

```
    # Sort rows based on absolute value of effect_var
    newframe = oldframe.iloc[oldframe[effect_var].abs().argsort()]

    # Now apply a function to each row of feature name to modify the name
    newframe[name_var] = newframe[name_var].apply(lambda x:
                                                  re.search(r'\((.+?),', x).group(1)+'_
                                                 if re.search(r'\((.+?),', x) else x)

    return newframe

LPM_df = pd.DataFrame()
LPM_df = pd.concat([LPM_Inference.params, LPM_Inference.conf_int(alpha = confidence_lev
LPM_df.columns = ['Feature', 'Effect', 'CI_low', 'CI_high']
LPM_df['Errors'] = (LPM_df['CI_high'] - LPM_df['CI_low'])/2.
LPM_df = LPM_df[LPM_df.Feature != 'Intercept']
LPM_df.drop(['CI_low', 'CI_high'], axis = 1, inplace = True)
LPM_df = modify_frame(LPM_df, 'Effect', 'Feature')


print('Features with 10 largest contributions to exit decisions are:\n')
LPM_largest = LPM_df['Feature'][-10:].to_list()
[LPM_largest[len(LPM_largest)-1-item] for item in np.arange(len(LPM_largest))]
```

```
Features with 10 largest contributions to exit decisions are:
```

Out[21]:
```
['JobRole_DirResearch',
 'JobRole_HealthRep',
 'OverTime_Yes',
 'JobRole_DirManufac',
 'JobRole_Scientist',
 'JobRole_Manager',
 'BusinessTravel_Frequent',
 'EducationField_HR',
 'Department_R&D',
 'EducationField_Technical']
```

The analyses above shows that features like OverTime, JobRole, BusinessTravel, etc, have the largest conribution to exit decisions of employees. The important thing to remeber here is that the effect could either be positive or negative. For example, doing 'OverTime' might incentivize employees to leave more. However, when someone reaches a position like Director of Research or Manufacturing then they are more likely to stay back.

In [22]:
```
# Now plot a graph

# fig, ax = plt.subplots(figsize = (82, 40))
# LPM_df.plot(x = 'Feature', y = 'Effect', yerr = 'Errors', kind = 'bar', color = 'none
#             ax = ax, ecolor = 'royalblue', fontsize = 20)
# ax.scatter(x = pd.np.arange(LPM_df.shape[0]), marker = 'o', s = 300, y = LPM_df['Effe
# ax.axhline(y = 0., linestyle = '--', color = 'red', linewidth = 5)
# ax.set_xticklabels(LPM_df['Feature'], fontsize = 50)
# ax.tick_params(axis = 'y', labelsize = 50)
# ax.set_xlabel('')
# ax.text(0, 0.375, 'Standard error spikes represent 95% CIs', fontsize = 60)
# plt.suptitle('Effects of Features on Probability of Attrition in Linear Probability M
```

(D) Logistic Regression Model to estimate Feature Importance in Explaining Exit Decision

Next, we implement a simple logistic regression model to estimate the effects of each feature (or levels of categorical variables) on the probability of attrition. Since logistic regression model is a

non-linear model, we have to estimate the marginal effects of each feature (or levels or categorical variables) and then we can compare them to the estimates from Linear Probability Model (LPM).

In [23]:
```
IBMData_numscaled_dummied = pd.get_dummies(IBMData_numscaled, drop_first = False)
IBMData_numscaled_dummied.drop([
    str(key)+'_'+re.search(r'"(.+?)"', val).group(1) if type(val) is str else str(key)+
    for key, val in baselevel_dict.items()], axis = 1, inplace = True)
#print(IBMData_numscaled_dummied.shape)
```

In [24]:
```
Logit_Inference = LogisticRegression(penalty = 'none').fit(IBMData_numscaled_dummied.dr
```

In [25]:
```
# Now we compute the marginal effects of the variables using the derivative method

Logit_df = pd.DataFrame()
a = np.sum(IBMData_numscaled_dummied.drop(['Attrition'], axis = 1)*Logit_Inference.coef
Logit_df['Effect'] = np.mean(np.exp(-a)/(1. + np.exp(-a))**2)*Logit_Inference.coef_[0]
Logit_df['Feature'] = IBMData_numscaled_dummied.drop(['Attrition'], axis = 1).columns.t
Logit_df = modify_frame(Logit_df, 'Effect', 'Feature')

print('Features with 10 largest contributions to exit decisions are:\n')
Logit_largest = Logit_df['Feature'][-10:].to_list()
[Logit_largest[len(Logit_largest)-1-item] for item in np.arange(len(Logit_largest))]
```

```
Features with 10 largest contributions to exit decisions are:
```

Out[25]:
```
['JobRole_DirResearch',
 'JobRole_HealthRep',
 'JobRole_DirManufac',
 'JobRole_Manager',
 'JobRole_Scientist',
 'OverTime_Yes',
 'BusinessTravel_Frequent',
 'Department_R&D',
 'JobRole_SalexExec',
 'JobRole_LabTech']
```

Again, the list above shows that features which have the largest contributions to exit decisions are OverTime, JobRole, BusinessTravel, etc. Similar to the LPM above, the effects could either be positive or negative.

In [26]:
```
# fig, ax = plt.subplots(figsize = (82, 40))
# Logit_df.plot(x = 'Feature', y = 'Effect', kind = 'bar', color = 'none', ax = ax, fon
# ax.scatter(x = pd.np.arange(Logit_df.shape[0]), marker = 'o', s = 300, y = Logit_df['
# ax.axhline(y = 0., linestyle = '--', color = 'red', linewidth = 5)
# ax.set_xticklabels(Logit_df['Feature'], fontsize = 50)
# ax.tick_params(axis = 'y', labelsize = 50)
# ax.set_xlabel('')
# ax.set_ylim(-0.5, 0.4)
# ax.text(0, 0.375, 'Standard error spikes represent 95% CIs', fontsize = 60)
# plt.suptitle('Effects of Features on Probability of Attrition in Logistic Regression
```

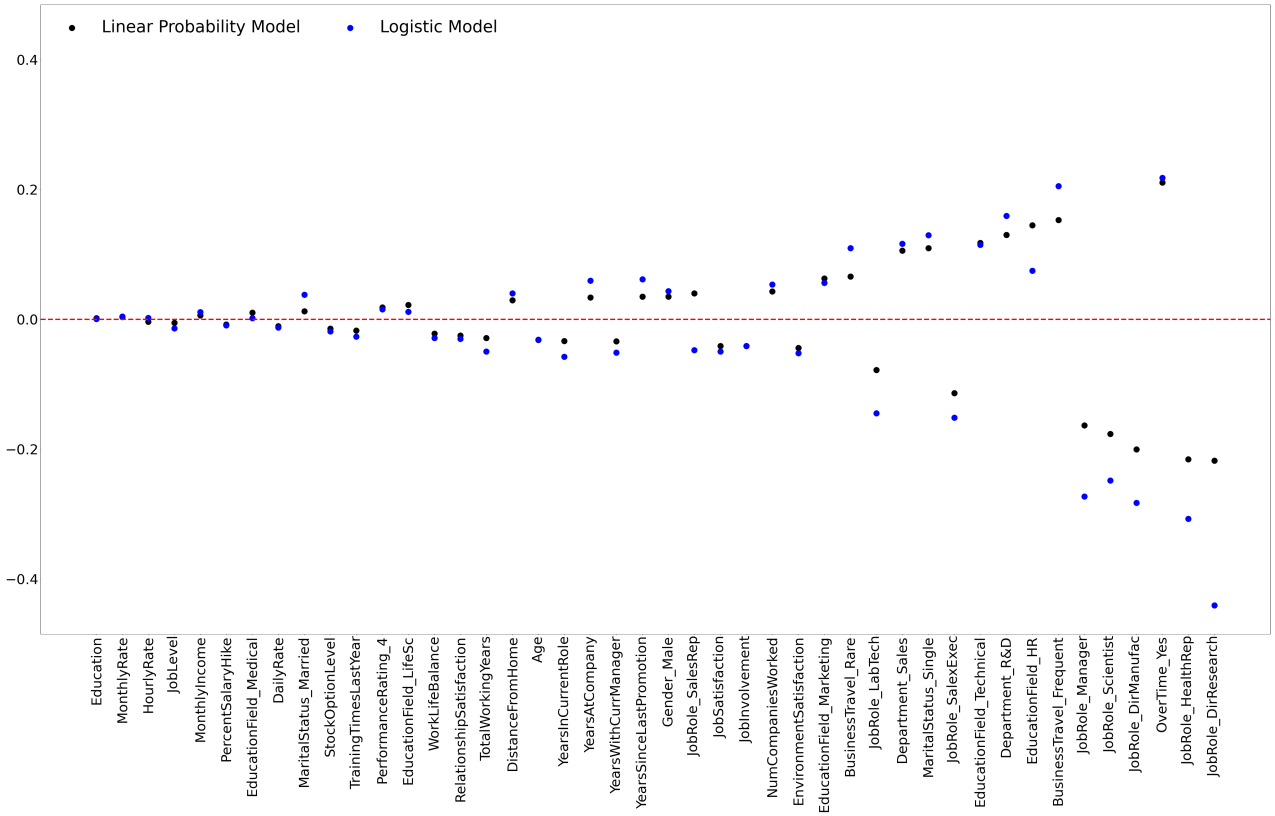
In [27]:
```
LPMLogit_df = LPM_df.drop(['Errors'], axis = 1).merge(Logit_df, on = 'Feature', suffixe
LPMLogit_df = LPMLogit_df.iloc[LPMLogit_df.Effect_LPM.abs().argsort()]

maxlimit = max(max(LPMLogit_df.Effect_LPM.abs()), max(LPMLogit_df.Effect_Logit.abs()))
```

(E) Comparison of Feature Importance in Explaining Exit Decisions between Linear and Logistic Regression Models

Finally, we compare the effect of each feature on the exit decision between linear and logistic regression models. To do this, we plot the features by the size of their 'absolute' effects on the exit decision in the Linear Regression model from left to right, with the rightmost feature having the largest (either positive or negative) effect on exit. Then, in the same graph we plot the marginal effects from the Logistic Regression model.

In [28]:
```
fig,ax = plt.subplots(figsize = (80,42))
ax.scatter(x = LPMLogit_df['Feature'], marker = 'o', s = 500, y = LPMLogit_df['Effect_L
ax.scatter(x = LPMLogit_df['Feature'], marker = 'o', s = 500, y = LPMLogit_df['Effect_L
ax.axhline(y = 0., linestyle = '--', color = 'red', linewidth = 5)
ax.set_xticklabels(LPMLogit_df['Feature'], fontsize = 50, rotation = 90)
ax.tick_params(axis = 'y', labelsize = 50)
ax.set_xlabel('')
ax.set_ylim(-maxlimit-maxlimit/10., maxlimit+maxlimit/10.)
ax.legend(loc = 'upper left', ncol = 2, prop={'size': 60})
plt.suptitle('Effects of Features on Probability of Attrition in Regression Models', fo
```

Effects of Features on Probability of Attrition in Regression Models



The plot above shows that the estimates are very close for both LPM and Logistic Models. This implies that features which are important determinants of Attrition in LPM are also important determinants in a Logistic Model.

(Step 3) Predicting Attrition Using Different Classifier Models

In this section, I predict attrition using 8 different classifier models by using data on features. To do this I first split the data into training (or training + validation) set and a test set. I train each model separately on the same training set and then predict attrition in the test set. Then I compare these 8

classifier models based on prediction metrics like accuracy, roc-auc score, precision, recall, f1 score, and the ROC Curve.

Classifier Models Used:

- (A) Logistic Regression (No Regularization)
- (B) Logistic Regression (l2 Regularization and Default penalty)
- (C) Logistic Regression (With Hyperparameter Tuning)
- (D) Gradient Boosting (Default Hyperparameters)
- (E) Gradient Boosting (With Hyperparameter Tuning)
- (F) Random Forest (Default Hyperparameters)
- (G) Random Forest (With Hyperparameter Tuning)
- (H) Neural Networks (in Tensorflow)

Metrics Used:

- (1) Accuracy
- (2) Precision
- (3) Recall
- (4) F1 Score
- (5) ROC-AUC Score
- (6) ROC Curve

First, we split the data into a training set (or trainining + validation set) and a test set. The split is 7:3:3.

In [29]:
```
# Split the data into train, validation, and test sets

IBMData_scaled = pd.get_dummies(IBMData[numvars + ordvars + catvars], drop_first = Fals
print('Shape of dummied data: {}'.format(IBMData_scaled.shape))
IBMData_scaled = pd.DataFrame(scaler.fit(IBMData_scaled).transform(IBMData_scaled),
                              columns = IBMData_scaled.columns)
print('Shape of dummied and scaled data: {}'.format(IBMData_scaled.shape))
IBMData_label = IBMData['Attrition']
print('Shape of labels: {}'.format(IBMData_label.shape))

seed = 59
train_prop = 0.70
val_prop = 0.15
test_prop = 0.15
Xtrainval, Xtest, Ytrainval, Ytest = train_test_split(IBMData_scaled, IBMData_label, te
Xtrain, Xval, Ytrain, Yval = train_test_split(Xtrainval, Ytrainval, test_size = val_pro

print('\nShapes of Generated Sets:')
print('Xtrain:', Xtrain.shape, 'Ytrain:', Ytrain.shape)
print('Xval:', Xval.shape, 'Yval:', Yval.shape)
print('Xtest:', Xtest.shape, 'Ytest:', Ytest.shape)
print('\nProportion of exits in:')
print('Ytrain: {:1.4}'.format(sum(Ytrain)/len(Ytrain)),
      'Yval: {:1.4}'.format(sum(Yval)/len(Yval)),
      'Ytest: {:1.4}'.format(sum(Ytest)/len(Ytest)))
```

```
Shape of dummied data: (1470, 52)
Shape of dummied and scaled data: (1470, 52)
Shape of labels: (1470,)
```

```
Shapes of Generated Sets:
Xtrain: (1028, 52) Ytrain: (1028,)
Xval: (221, 52) Yval: (221,)
Xtest: (221, 52) Ytest: (221,)

Proportion of exits in:
Ytrain: 0.1634 Yval: 0.1629 Ytest: 0.1493
```

In [30]:

```
# Create Storages for the variables

global test_accuracy, test_precision, test_recall, test_rocauc, test_roccurve, train_ac
test_accuracy = []
test_precision = []
test_recall = []
test_rocauc = []
test_f1 = []
test_roccurve = []
train_accuracy = []
train_precision = []
train_recall = []
train_rocauc = []
train_f1 = []
train_roccurve = []

# Write a function which takes the predicted labels or probabilities and appends the me
def compute_predictionmetrics(actuallabel, predictedlabel, predictedprob, traintesttype
    if(include):
        if(traintesttype == 'train'):
            train_accuracy.append(accuracy_score(actuallabel, predictedlabel))
            train_precision.append(precision_score(actuallabel, predictedlabel))
            train_recall.append(recall_score(actuallabel, predictedlabel))
            train_f1.append(f1_score(actuallabel, predictedlabel))
            train_rocauc.append(roc_auc_score(actuallabel, predictedlabel))

            train_roccurve.append(roc_curve(actuallabel, predictedprob))
        else:
            test_accuracy.append(accuracy_score(actuallabel, predictedlabel))
            test_precision.append(precision_score(actuallabel, predictedlabel))
            test_recall.append(recall_score(actuallabel, predictedlabel))
            test_f1.append(f1_score(actuallabel, predictedlabel))
            test_rocauc.append(roc_auc_score(actuallabel, predictedlabel))

            test_roccurve.append(roc_curve(actuallabel, predictedprob))
    if(draw):
        print('Test Set Prediction Metrics:')
        print('Accuracy: {:1.4}'.format(accuracy_score(actuallabel, predictedlabel)))
        print('Precision: {:1.4}'.format(precision_score(actuallabel, predictedlabel)))
        print('Recall: {:1.4}'.format(recall_score(actuallabel, predictedlabel)))
        print('F1: {:1.4}'.format(f1_score(actuallabel, predictedlabel)))
        print('ROCAUC: {:1.4}'.format(roc_auc_score(actuallabel, predictedlabel)))

        fpr, tpr, _ = roc_curve(actuallabel, predictedprob)
        plt.figure()
        plt.plot(fpr, tpr)
        plt.plot([0, 1], [0, 1], 'r--')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
```

```
        plt.title('Receiver operating characteristic')
        plt.show()
```

(A) Logistic Regression Classifier (with No Regularization)

In [31]:
```
# Fit the model to the train and validation data

Logit = LogisticRegression(penalty = 'none').fit(Xtrainval, Ytrainval)
```

In [32]:
```
# Predict the labels in test set and then call the functions
Ypred_Logit = Logit.predict(Xtest)

# Now call the function for plotting and recording metrics
compute_predictionmetrics(Ytest, Ypred_Logit, Logit.predict_proba(Xtest)[:,1], 'test',
compute_predictionmetrics(Ytrainval, Logit.predict(Xtrainval), Logit.predict_proba(Xtra
```

(B) Logistic Regression with Regularization (l2 norm and default regularization parameter)

In [33]:
```
Logit_regular = LogisticRegression(penalty = 'l2', C = 1.0).fit(Xtrainval, Ytrainval)
```

In [34]:
```
# Predict the labels in test set and then call the functions
Ypred_Logit_regular = Logit_regular.predict(Xtest)

# Now call the function for plotting and recording metrics
compute_predictionmetrics(Ytest, Ypred_Logit_regular, Logit_regular.predict_proba(Xtest
compute_predictionmetrics(Ytrainval, Logit_regular.predict(Xtrainval), Logit_regular.pr
```

(C) Logistic Regression Classifier with Hyperparameter Tuning

In [35]:
```
# Define the set of parameters which need to be iterated over
param_grid = {'penalty':['l1','l2'],
              'C':np.logspace(-4, 4, 50),
              'fit_intercept':[True, False]
              }

Logit_tuned = GridSearchCV(LogisticRegression(), param_grid, cv = 3, verbose = 0, scori
```

In [36]:
```
Ypred_Logit_tuned = Logit_tuned.predict(Xtest)

# Now show the test prediction metrics and record them
compute_predictionmetrics(Ytest, Ypred_Logit_tuned, Logit_tuned.predict_proba(Xtest)[:,
compute_predictionmetrics(Ytrainval, Logit_tuned.predict(Xtrainval), Logit_tuned.predic
```

In [37]:
```
# Show the optimal parameters

print('Optimized Parameters for Logistic Regression Classifier: ')
Logit_tuned.best_params_
```

```
Optimized Parameters for Logistic Regression Classifier:
```

Out[37]:
```
{'C': 0.08685113737513521, 'fit_intercept': True, 'penalty': 'l2'}
```

(D) Gradient Boosting Classifier (with default hyperparameters)

In [38]:
```
GBC = GradientBoostingClassifier(n_estimators = 100,
                                 learning_rate = 0.01,
                                 max_depth = 50,
                                 random_state = seed,
                                 max_features = 'auto',
```

```
                                    verbose = 0,
                                    validation_fraction = val_prop/(1.-test_prop)).fit(Xtra
```

In [39]:
```
Ypred_GBC = GBC.predict(Xtest)

compute_predictionmetrics(Ytest, Ypred_GBC, GBC.predict_proba(Xtest)[:,1], 'test', draw
compute_predictionmetrics(Ytrainval, GBC.predict(Xtrainval), GBC.predict_proba(Xtrainva
```

(E) Gradient Boosting Classifier with Hyperparameter Tuning

In [40]:
```
param_grid = {'n_estimators':np.arange(100, 510, 100),
              'learning_rate':[0.001, 0.005, 0.01, 0.05, 0.1],
              'max_depth':np.arange(10, 110, 20),
              'max_features':['auto','log2', None]}

GBC_tuned = GridSearchCV(GradientBoostingClassifier(random_state = seed,
                                                    verbose = 0,
                                                    validation_fraction = val_prop,
                                                   criterion = 'friedman_mse'),
                         param_grid, cv = 3, verbose = 0, scoring = 'accuracy').fit(Xtr
```

In [41]:
```
Ypred_GBC_tuned = GBC_tuned.predict(Xtest)

compute_predictionmetrics(Ytest, Ypred_GBC_tuned, GBC_tuned.predict_proba(Xtest)[:,1],
compute_predictionmetrics(Ytrainval, GBC_tuned.predict(Xtrainval), GBC_tuned.predict_pr
```

In [42]:
```
print('Optimized Hyperparameters of Gradient Boosting Classifier:')
GBC_tuned.best_params_
```

```
Optimized Hyperparameters of Gradient Boosting Classifier:
```

Out[42]:
```
{'learning_rate': 0.05,
 'max_depth': 10,
 'max_features': 'log2',
 'n_estimators': 300}
```

(F) Random Forest Classifier (with default hyperparameters)

In [43]:
```
RF = RandomForestClassifier(n_estimators = 100,
                            max_depth = 50,
                            random_state = seed,
                            max_features = 'auto',
                            verbose = 0,
                            bootstrap = True,
                            oob_score = True).fit(Xtrainval, Ytrainval)
```

In [44]:
```
Ypred_RF = RF.predict(Xtest)

compute_predictionmetrics(Ytest, Ypred_RF, RF.predict_proba(Xtest)[:,1], 'test', draw =
compute_predictionmetrics(Ytrainval, RF.predict(Xtrainval), RF.predict_proba(Xtrainval)
```

(G) Random Forest Classifier with hyperparameter optimization

In [45]:
```
param_grid = {'n_estimators':np.arange(100, 510, 100),
              'max_depth':np.arange(10, 110, 20),
              'max_features':['auto','log2',None]}

RF_tuned = GridSearchCV(RandomForestClassifier(random_state = seed,
                                               verbose = 0,
                                               bootstrap = True,
```

```
                                                oob_score = True),
                           param_grid, cv = 3, verbose = 0, scoring = 'accuracy').fit(Xtrai
```

In [46]:
```
Ypred_RF_tuned = RF_tuned.predict(Xtest)

compute_predictionmetrics(Ytest, Ypred_RF_tuned, RF_tuned.predict_proba(Xtest)[:,1], 't
compute_predictionmetrics(Ytrainval, RF_tuned.predict(Xtrainval), RF_tuned.predict_prob
```

In [47]:
```
print('Optimized Hyperparameters of Random Forest Classifier:')
RF_tuned.best_params_
```

```
Optimized Hyperparameters of Random Forest Classifier:
```

Out[47]:
```
{'max_depth': 10, 'max_features': 'auto', 'n_estimators': 400}
```

In [48]:
```
# metrics_list = [test_accuracy, test_precision, test_recall, test_f1, test_rocauc, tes
#                 train_accuracy, train_precision, train_recall, train_f1, train_rocauc

# with open(os.path.join(sourcepath, 'reports', 'PredictionMetrics.txt'), 'w') as f:
#     write = csv.writer(f)
#     for item in metrics_list:
#         write.writerow(item)

# newlist = []
# with open(os.path.join(sourcepath, 'reports', 'PredictionMetrics.txt'), 'r') as r:
#     csv_reader = reader(r)
#     for row in csv_reader:
#         newlist.append(row)
```

(H) Neural Networks (Implemented in Tensorflow)

Our basic neural network has no hidden layer, but only an input layer with 128 units (neurons) and a single output neuron. The input layer has a relu activation function while the output layer as a sigmoid activation function. We train the NN either for 500 epochs or till the training accuracy achieved is 95%, whichever comes first. It is optimized using the 'adam' optimizer, the loss function is 'binary_crossentropy'.

In [101…
```
# Define the model first
NNC = tf.keras.models.Sequential([
    tf.keras.layers.Dense(input_shape = (Xtrainval.shape[1],), units = 1024, activation
    tf.keras.layers.Dense(units = 1, activation = tf.nn.sigmoid)
])
print('Basic Neural Network Architecture:')
print(NNC.summary())

# Compile the model
NNC.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 0.0005), loss = 'bi

# Define a callback function which will terminate training when a certain condition is
accuracy_threshold = 0.95
num_epochs = 500

global final_epoch
final_epoch = 0
class myCallBack(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs = {}):
        if(logs.get('acc') > accuracy_threshold):
            print('Desired Accuracy Threshold Reached. Aborting Training. Epoch reached
            global final_epoch
```

```
            final_epoch = epoch
            self.model.stop_training = True
callbacks = myCallBack()

# Train the model
history = NNC.fit(Xtrainval, Ytrainval, epochs = num_epochs, callbacks = [callbacks], v
```

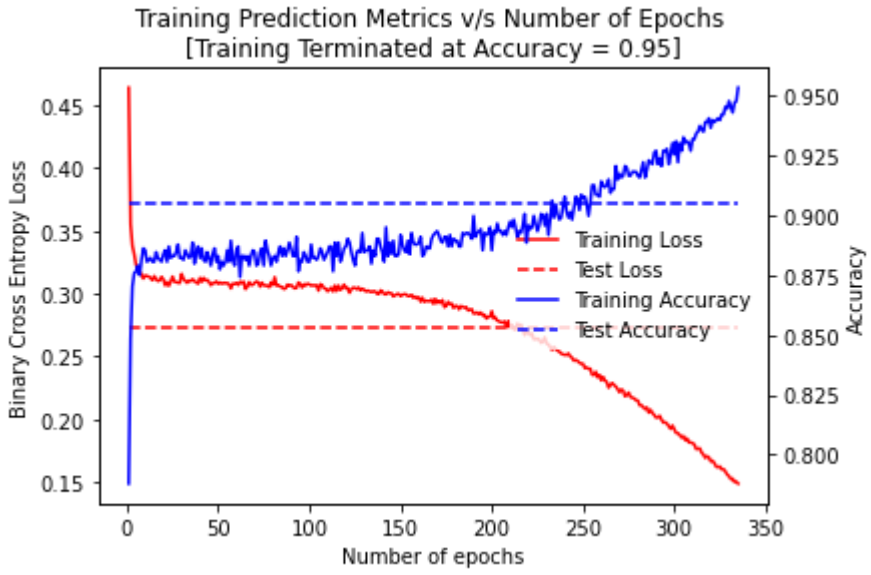
```
Basic Neural Network Architecture:
Model: "sequential_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_30 (Dense)             (None, 1024)              54272
_________________________________________________________________
dense_31 (Dense)             (None, 1)                 1025
=================================================================
Total params: 55,297
Trainable params: 55,297
Non-trainable params: 0
_________________________________________________________________
None
Desired Accuracy Threshold Reached. Aborting Training. Epoch reached: 334
```

In [102…

```
test_loss_NNC, test_accuracy_NNC = NNC.evaluate(Xtest, Ytest, verbose = 0)
```

In [103…

```
if final_epoch == 0:
    final_epoch = len(history.history['acc'])-1
fig, ax = plt.subplots()
p1 = ax.plot(np.arange(1, final_epoch + 2), history.history['loss'],
             color = 'red', label = 'Training Loss')
p2 = ax.plot(np.arange(1, final_epoch + 2), [test_loss_NNC for _ in np.arange(1, final_
             color = 'red', linestyle = 'dashed', label = 'Test Loss')
ax.set_title('Training Prediction Metrics v/s Number of Epochs \n[Training Terminated a
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Binary Cross Entropy Loss')
axd = ax.twinx()
p3 = axd.plot(np.arange(1, final_epoch + 2), history.history['acc'], color = 'blue', la
p4 = axd.plot(np.arange(1, final_epoch + 2), [test_accuracy_NNC for _ in np.arange(1, f
             color = 'blue', linestyle = 'dashed', label = 'Test Accuracy')
p = p1 + p2 + p3 + p4
labs = [l.get_label() for l in p]
ax.legend(p, labs, loc = 'center right')
axd.set_ylabel('Accuracy')
plt.show();
```



In [104…
```
Ypred_NNC = NNC.predict_classes(Xtest)

compute_predictionmetrics(Ytest, Ypred_NNC, NNC.predict_proba(Xtest), 'test', draw = Fa
compute_predictionmetrics(Ytrainval, NNC.predict_classes(Xtrainval), NNC.predict_proba(
```

In [105…
```
# global accuracy_threshold
# accuracy_threshold = 0.95
# global num_epochs
# num_epochs = 500
```

In [106…
```
# def create_and_run_model(hparams, Xtrain, Ytrain, Xval, Yval, Xtest, Ytest):

#     class myCallBack(tf.keras.callbacks.Callback):
#         def on_epoch_end(self, epoch, logs = {}):
#             if(logs.get('acc') > accuracy_threshold):
#                 #print('Desired Accuracy reached. Aborting Training at Epoch: {}'.for
#                 self.model.stop_training = True
#     callbacks = myCallBack()


#     model = tf.keras.models.Sequential([
#         tf.keras.layers.Dense(input_shape = (Xtrain.shape[1], ), units = hparams['num_
#         tf.keras.layers.Dense(units = 1, activation = hparams['activation_output'])
#     ])

#     optimizer = hparams['optimizer']
#     lr = hparams['lr']
#     if optimizer == 'adam':
#         optimizer = tf.keras.optimizers.Adam(learning_rate = lr)
#     elif optimizer == 'sgd':
#         optimizer = tf.keras.optimizers.SGD(learning_rate = lr)
#     else:
#         optimizer = tf.keras.optimizers.RMSprop(learning_rate = lr)
#     model.compile(optimizer = optimizer, loss = 'binary_crossentropy', metrics = ['ac

#     history = model.fit(Xtrain, Ytrain, validation_data = (Xval, Yval), epochs = num_

#     testaccuracy = accuracy_score(Ytest, model.predict_classes(Xtest))
#     testprecision = precision_score(Ytest, model.predict_classes(Xtest))
#     testrecall = recall_score(Ytest, model.predict_classes(Xtest))
```

```
#     testf1 = f1_score(Ytest, model.predict_classes(Xtest))
#     testrocauc = roc_auc_score(Ytest, model.predict_classes(Xtest))
#     testroccurve = roc_curve(Ytest, model.predict_proba(Xtest))

#     trainaccuracy = accuracy_score(Ytrain, model.predict_classes(Xtrain))
#     trainprecision = precision_score(Ytrain, model.predict_classes(Xtrain))
#     trainrecall = recall_score(Ytrain, model.predict_classes(Xtrain))
#     trainf1 = f1_score(Ytrain, model.predict_classes(Xtrain))
#     trainrocauc = roc_auc_score(Ytrain, model.predict_classes(Xtrain))
#     trainroccurve = roc_curve(Ytrain, model.predict_proba(Xtrain))

#     valaccuracy = history.history['val_acc'][-1]

#     return valaccuracy, (testaccuracy, testprecision, testrecall, testf1, testrocauc,
```

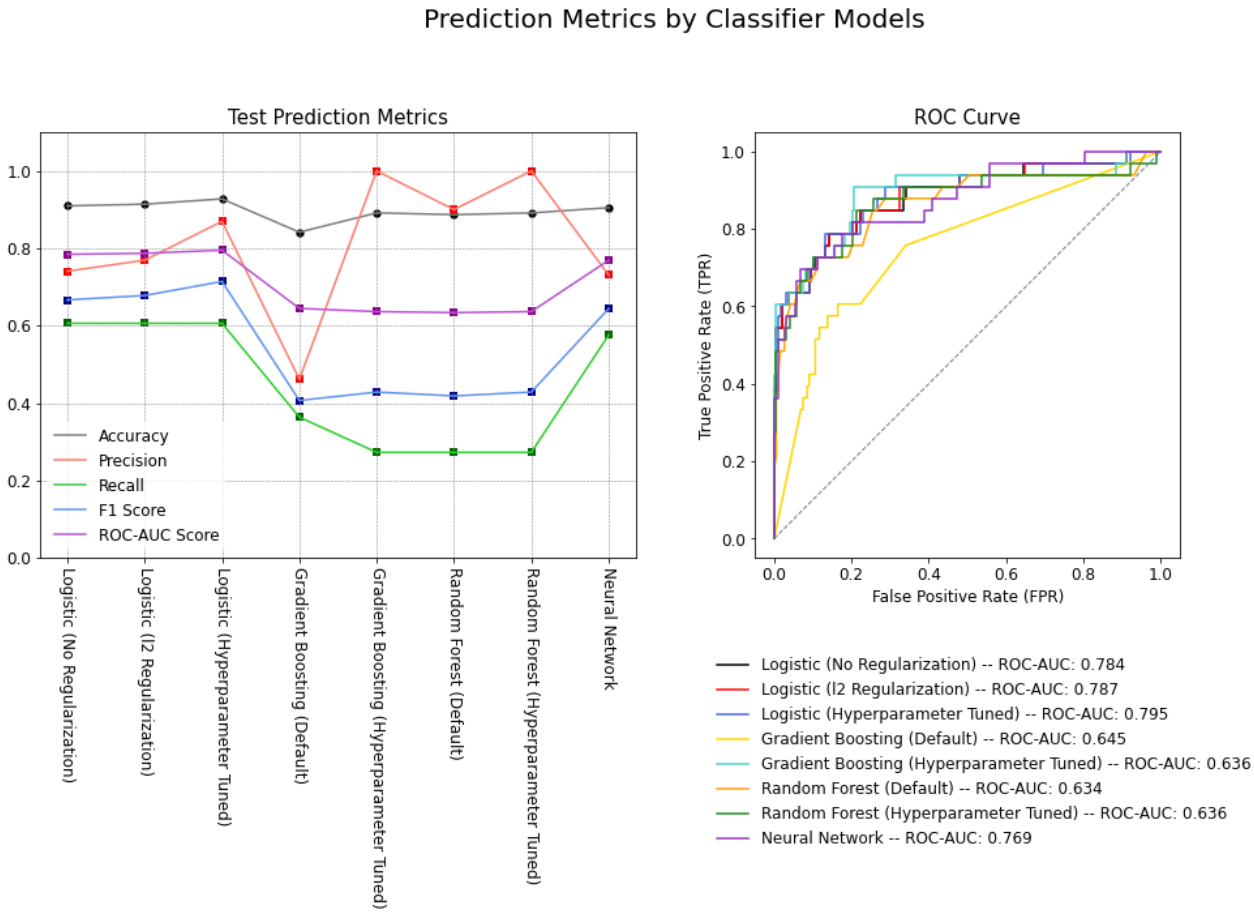
In [107…

```
# numunits = [128, 256]
# learningrate = [0.001, 0.005, 0.01]
# optimizermethod = ['adam', 'sgd', 'rmsprop']
# activationfunc = ['relu', 'sigmoid']

# tf.logging.set_verbosity(tf.logging.ERROR)
# counter = 0
# optimal_accuracy = 0.
# for units in numunits:
#     for lr in learningrate:
#         for optimizer in optimizermethod:
#             for activation_input in activationfunc:
#                 for activation_output in activationfunc:

#                     if(counter%10 == 0):
#                         print('Session Number: {}'.format(counter))
#                     counter = counter + 1
#                     # Create a dictionary to pass on hyperparameters
#                     hparams = {'num_units':units,
#                                'lr':lr,
#                                'optimizer':optimizer,
#                                'activation_output':activation_output,
#                                'activation_input':activation_input}
#                     #print(hparams)
#                     accuracyvalue, metrics = create_and_run_model(hparams, Xtrain, Yt
#                     if accuracyvalue > optimal_accuracy:
#                         optimal_accuracy = accuracyvalue
#                         optimal_metrics = copy.deepcopy(metrics)
#                         optimal_params = copy.deepcopy(hparams)
```

In [108…

```
model_list = ['Logistic (No Regularization)',
              'Logistic (l2 Regularization)',
              'Logistic (Hyperparameter Tuned)',
              'Gradient Boosting (Default)',
              'Gradient Boosting (Hyperparameter Tuned)',
              'Random Forest (Default)',
              'Random Forest (Hyperparameter Tuned)',
              'Neural Network']

fig = plt.subplots(figsize = (18,6))

ax1 = plt.subplot(1, 2, 1)
ax1.scatter(x = model_list, y = test_accuracy, marker = 'o', s = 40, color = 'black', l
plt.plot(model_list, test_accuracy, color = 'grey', label = 'Accuracy')
ax1.scatter(x = model_list, y = test_precision, marker = 's', s = 40, color = 'red', la
```

```
plt.plot(model_list, test_precision, color = 'salmon', label = 'Precision')
ax1.scatter(x = model_list, y = test_recall, marker = 's', s = 40, color = 'darkgreen',
plt.plot(model_list, test_recall, color = 'limegreen', label = 'Recall')
ax1.scatter(x = model_list, y = test_f1, marker = 's', s = 40, color = 'darkblue', labe
plt.plot(model_list, test_f1, color = 'cornflowerblue', label = 'F1 Score')
ax1.scatter(x = model_list, y = test_rocauc, marker = 's', s = 40, color = 'darkmagenta
plt.plot(model_list, test_rocauc, color = 'mediumorchid', label = 'ROC-AUC Score')
ax1.set_ylim([0., 1.1])
ax1.set_xticklabels(model_list, rotation = 270, fontsize = 12)
ax1.tick_params(labelsize = 12)
plt.grid(which = 'major', color = 'grey', linestyle = 'dashed', linewidth = 0.5)
plt.legend(loc = 'lower left', frameon = True, fancybox = True, fontsize = 12)
ax1.set_title('Test Prediction Metrics', fontsize = 15);

ax2 = plt.subplot(1, 2, 2)
ax2.plot(test_roccurve[0][0], test_roccurve[0][1], color = 'black',
         label = 'Logistic (No Regularization) -- ROC-AUC: {:1.3}'.format(test_rocauc[0
ax2.plot(test_roccurve[1][0], test_roccurve[1][1], color = 'red',
         label = 'Logistic (l2 Regularization) -- ROC-AUC: {:1.3}'.format(test_rocauc[1
ax2.plot(test_roccurve[2][0], test_roccurve[2][1], color = 'royalblue',
         label = 'Logistic (Hyperparameter Tuned) -- ROC-AUC: {:1.3}'.format(test_rocau
ax2.plot(test_roccurve[3][0], test_roccurve[3][1], color = 'gold',
         label = 'Gradient Boosting (Default) -- ROC-AUC: {:1.3}'.format(test_rocauc[3]
ax2.plot(test_roccurve[4][0], test_roccurve[4][1], color = 'mediumturquoise',
         label = 'Gradient Boosting (Hyperparameter Tuned) -- ROC-AUC: {:1.3}'.format(t
ax2.plot(test_roccurve[5][0], test_roccurve[5][1], color = 'orange',
         label = 'Random Forest (Default) -- ROC-AUC: {:1.3}'.format(test_rocauc[5]))
ax2.plot(test_roccurve[6][0], test_roccurve[6][1], color = 'forestgreen',
         label = 'Random Forest (Hyperparameter Tuned) -- ROC-AUC: {:1.3}'.format(test_
ax2.plot(test_roccurve[7][0], test_roccurve[7][1], color = 'darkorchid',
         label = 'Neural Network -- ROC-AUC: {:1.3}'.format(test_rocauc[7]))
ax2.plot(np.arange(0., 1.1, 0.1), np.arange(0., 1.1, 0.1), color = 'slategrey', linesty
ax2.set_aspect('equal', anchor = 'SW')
ax2.tick_params(labelsize = 12)
ax2.set_xlabel('False Positive Rate (FPR)', fontsize = 12)
ax2.set_ylabel('True Positive Rate (TPR)', fontsize = 12)
plt.legend(bbox_to_anchor=(1.2, -0.2), fontsize = 12)
ax2.set_title('ROC Curve', fontsize = 15)
plt.suptitle('Prediction Metrics by Classifier Models', fontsize = 20, y = 1.1)

plt.savefig(os.path.join(sourcepath, 'reports', 'figures', 'PredictionMetricsComparison
            bbox_inches = 'tight')
```



Conclusion:

- Best model from among the 8 classifier models: Logistic Regression with Tuned Hyperparameters
- Best Model Prediction Metrics:
  - Accuracy: 0.93
  - Precision: 0.87
  - Recall: 0.61
  - F1-Score: 0.71
  - ROC-AUC Score: 0.80
- Best Model Parameters:
  - Penalty: 'l2'
  - C: 0.08685
  - fit_intercept = True
- Logistic Regression with Hyperparameter Tuning achieves 2% increase in test accuracy over simple Logistic Regression Classifier
- Logistic Regression with Hyperparameter Tuning achieves 2.5% increase in test accuracy over simple Neural Network Classifier